# R-SHT: a State History Tree with R-Tree Properties for Analysis and Visualization of Highly Parallel System Traces

LOÏC PRIEUR-DREVON, Polytechnique Montréal
RAPHAËL BEAMONTE, Polytechnique Montréal
MICHEL R. DAGENAIS, Polytechnique Montréal

Understanding the behaviour of large computer systems with many threads and cores is a challenging task. Dynamic analysis tools such as tracers have been developed to assist programmers in debugging and optimizing the performance of such systems. However, complex systems can generate huge traces, with billions of events, which are hard to analyze manually. Trace visualization and analysis programs aim to solve this problem. Such software needs fast access to data, which a linear search through the trace cannot provide. Several programs have resorted to stateful analysis to rearrange data into more query friendly structures.

In previous work, we suggested modifications to the State History Tree (SHT) data structure to correct its storage and memory usage. While the improved structure, eSHT, made near optimal external memory usage and had internal reduced memory usage, we found that query performance, while twice as fast, exhibited scaling limitations.

In this paper, we proposed a new structure using R-Tree techniques to improve query performance. We explain the hybrid scheme and algorithms used to optimize the structure to model the expected behaviour. Finally, we benchmark the data structure on highly parallel traces and on a demanding trace visualization use case.

Our results show that the hybrid R-SHT structure retains the eSHT's optimal disk usage properties while providing several orders of magnitude speed up to queries on highly parallel traces.

CCS Concepts: •**Information systems** → **Multidimensional range search; B-trees;** •**Theory of computation** → **Sorting and searching;** •**Software and its engineering** → *State systems; Dynamic analysis;* Massively parallel systems;

Additional Key Words and Phrases: Data Structures, Tree, Stateful Analysis

**ACM Reference Format:**
Loïc Prieur-Drevon, Raphaël Beamonte, Michel R. Dagenais, 2017. R-SHT: a State History Tree with R-Tree Properties for Analysis and Visualization of Highly Parallel System Traces *ACM Trans. Model. Perform. Eval. Comput. Syst.* X, Y, Article Z (June 2016), 29 pages.
DOI: 0000001.0000001

## 1. INTRODUCTION

Understanding the runtime behavior of complex computer systems is a daunting task. Tracing is one of many runtime analysis methods used to instrument and collect data on systems and applications. Compared to logging, tracers have much lower overhead and can produce hundreds of thousands of events per second, at nanosecond precision, providing extremely detailed information on kernel, process and hardware states.

Tracers produce trace files, a series of chronological events, which are optimized for low overhead and data storage but challenging for human operators to understand. A number of software solutions, called trace visualizers, have been developed to facilitate the understanding of these files by providing graphical visualizations, statistics and detailed analysis of certain use cases. These programs perform stateful analysis, that transform event-based data structures into state-based structures, and reorganize data from lists to trees, for faster access.

Indeed, when traces reach gigabyte or terabyte size, efficient data structures are important for maintaining sustainable performance levels for analysis, and low latency for interactive visualizations. Said data structures must be able to scale horizontally – for tracing programs over a large duration – as well as vertically – for tracing systems with many processors, threads and resources.

Among the existing data structures, some are optimized for disk storage, build time or perhaps query performance. When working on trace visualization, the latter is fairly important. R-Trees are a family of data structures used to index multi-dimensional data sets and offer excellent query performance.

In previous work [Prieur-Drevon et al. 2016], we presented a self-defined tree structure, optimized for external memory storage and with satisfactory query performance. However, we found that query performance scaled linearly to the number of components in the system, which led to slowdowns for the analysis of systems with many threads for example.

In this paper, we propose an enhanced, configurable build algorithm that reorganizes data in the sub-trees so that they reflect properties of an efficient R-Tree.

This paper is organized as follows. First we cover related research on trace visualizers and underlying data structures in section 2. Then we present the architecture of the current data structure in section 3 as well as that of the evolutions we suggest in sections 4 and 5. In section 6, we model the behavior of the query algorithms before benchmarking them on real-life traces in section 7. Finally, we conclude and suggest future work.

## 2. RELATED WORK

### 2.1. Trace visualizers

In this section, we compare open source trace visualizers that deal with stateful analysis and have a documented data structure to store this information.

`Jumpshot` [Chan et al. 2008] is the visualization component for the MPI Parallel Environment software package. It displays the nodes' states evolutions over time and the messages that they have exchanged. Jumpshot uses the `slog2` format to reduce the cost of accessing trace data. When using the MPE tracing framework for MPI, users have the option for a state based logging format, in which the tracer directly produces state intervals, as opposed to event based tracing, which produces a list of timestamped events. However, Jumpshot is focused on MPI visualization and doesn't provide detailed analysis capabilities.

`Paraprof` [Geimer et al. 2007] uses tracing and profiling techniques to summarize information, allowing it to scale well to HPC applications. It stores data in a `CUBE` [Geimer et al. 2007] data structure, which is based around a Cube data model, with one dimension for metrics, another for programs and a third dimension for the system. When in memory, Paraprof stores its data as a double level map of vectors, keyed by the metric, then the call path and finally the process number. Despite all its capabilities, Paraprof does not provide stateful information on the systems' performance, rather focusing on metrics.

`Aftermath` [Pop and Cohen 2013] provides visualization and analysis for traces from task-parallel work-flows. As part of the OpenStream project, it relies heavily on aggregation of trace points from the application as well as the runtime, and performance counters. Its creators state that the software can scale up to traces of several gigabytes in size while

remaining fast thanks to the use of augmented interval trees as a backend. However, Aftermath stores the entire trace in memory, thus limiting its scalability.

Google has built tracing into `Chromium` [Google 2013] to help developers identify slowdowns originating from either JavaScript, C++, or other bottlenecks. The visualizer easily scales to the number of threads used by chrome and the flame-graphs of some deep call stacks. Withal, Chromium Tracing is obviously restricted to analyzing Chrome's performance, yet shows the appeal of tracing and analysis for diversified applications.

`Pajé ViTE`[Coulomb et al. 2012] is developed for Pajé or OTF traces from parallel or distributed applications. It can scale to display millions of events per view and large computing clusters by storing trace events in a balanced binary tree, which is however limited by the size of the main memory.

`Trace Compass` [Côté and Dagenais 2016] is the extensible trace visualizer and analyzer for traces generated by the `LTTng` [Desnoyers and Dagenais 2006b] tracer and other tracing tools. It is built using the Eclipse framework and uses State History Trees (SHT) to store state data in a query-efficient structure. It supports a number of different trace formats and offers comprehensive analysis modules. Because of its flexibility, it is equally effective for analysing real-time programs running on a single system, as it is with multi-threading, DSP and GPU architectures, and distributed or virtualized systems.

Distributed systems, which rely on the MPI standard also have a number of dedicated tools to analyse their specificities.

`HPCTraceviewer` is the visualization component in the HPCToolkit, it is used for performance measurement and analysis on large supercomputers. By relying on a client/server architecture, it avoids moving gigabytes of trace files and benefits from the computing power and memory of MPI nodes to process raw data.

The `VampirTrace` [Müller et al. 2007] visualizer relies on a client/server architecture with parallel servers to scale up to reading large distributed traces. The nodes interact via standard MPI primitives and precompute the required information before sending the results over to the client.

`ScalaTrace` [Noeth et al. 2009] relies on local and global compression to reduce the sizes of MPI traces dramaticaly and preprocess trace comparison. This results in constant size or sublinear growth sizes compared to the number of nodes.

However, when working on huge traces, the aforestated software cannot afford to query directly the trace itself, as the query length could grow linearly with the trace size. This is why such programs transform traces into other data structures that are more efficient for querying. Most programs choose to store "stateful" data, i.e., one object per state [Ezzati-Jivan and Dagenais 2012]. For example, the state of the Attribute "thread/42/Status" could be "Sleep" between two specific time-stamps.

### 2.2. Stateful Data Structures

In this section, we compare the data structures used by aforementioned trace visualizers and generic data structures used for multidimensional data. The following structures focus on query performance.

`B-Trees` [Comer 1979] were one of the first index structures developed to accelerate accesses to external memory data structures. B-Trees extend binary search trees by giving each node between $d$ and $2d$ keys as well as $d+1$ to $2d+1$ pointers to children nodes, in which case the tree is of `order d`. All the values in the sub-tree referenced by the $i^{th}$ pointer are larger than the $i^{th}$ key and smaller than the $(i+1)^{th}$.

`Multi-version B-Trees` [Becker et al. 1996] store data items of the type $< key, t_{start}, t_{end}, pointer >$ where $key$ is unique for every version and $t_{start}, t_{end}$ are the version numbers for the item's lifespan. It has a number of B-Tree root nodes that each stand for an interval of versions. Each operation (insertion or deletion) creates a new ver-

sion. Versioning uses live blocks which duplicate the open intervals of the old block and have free space to store future values.

`Interval Trees` [Cormen 2009] are tree structures designed to efficiently find time intervals that overlap a certain timestamp. Different implementations of interval trees exist in the literature. `Aftermath` for example, uses `Augmented Interval Trees` [Har-Peled 2011] which are based on ordered tree structures. These are typically binary trees or self-balancing binary search trees, where the interval start time is used for ordering. Each node is "augmented" with the latest end time of the associated sub-tree. Knowing the end times of the sub-tree tells the algorithms which nodes they can skip when searching for intervals. The `Centered Interval Tree` implementation is similar to a binary search tree, with each node using a time $t$ as a key such that all the intervals in the left node end before $t$, all the intervals in the right node start after $t$, and the node contains all intervals overlapping $t$. The tree is balanced when the left and right sub-trees contain a similar number of intervals. `Segment Trees` [Berg et al. 2008] are especially efficient for retrieving segments that overlap a certain value. Segment Trees are based on binary search trees, with nodes defined by the range they span, called "interval". Each segment may have several pointers in the tree, in the shallowest possible nodes, such that these nodes' intervals span the segment but that the parent's interval does not span the segment.

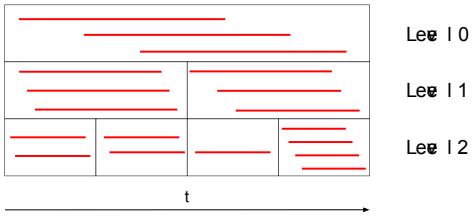



Fig. 1: Representation of the `slog2` data structure [Chan et al. 2008]

The `slog2` data structure [Chan et al. 2008] uses a balanced binary tree structure keyed by time. Each node is defined by a start and end time, such that children nodes' durations are half that of their parent's. Moreover, sibling nodes' times cannot overlap and the root node's duration is that of the trace. Intervals fit in the shortest node that can contain both their end and start time. As nodes and state intervals cannot overlap, the challenge is finding the right depth (leaf node length) to obtain a high fill ratio. Figure 1 shows a representation of that data structure.

The `State History Tree` (SHT) [Montplaisir-Goncalves et al. 2013] structure was designed with event-based trace analysis and visualization storage in mind. Stateful analysis results, in the form of state intervals: $< key, time_{start}, time_{end}, value >$, are stored in the tree in a single pass through the trace. The SHT is also designed to perform well on rotating media, so each node is mapped to a block on external memory. SHTs support the creation of new keys assigned to a state machine and the update of said states. SHT nodes are defined by their start and end times, such that all the intervals stored in a node must be included in these bounds. Moreover a child's bounds must be included in its parent's and cannot overlap between siblings. The SHT's construction begins with a single leaf node, siblings and parents are added as nodes are filled. The SHT's structure and limitations are further discussed in section 3.

### 2.3. R-Trees

R-Trees [Guttman 1984] are used to store multidimensional data. The points stored in each node are mandatorily included in the node's Minimum Bounding Rectangle (MBR), a hyper-rectangle which bounds the points on each dimension of the tree. Children nodes' MBRs are included in their parent's MBR. The challenge is to minimize the volume or overlap of these MBRs in order to limit the number of nodes that need to be searched during queries. Structuring the tree is usually done during the insertion phase. When inserting a new point, the algorithms select the best node into which insert new points, usually the ones with MBR that overlap the point, or require the least enlargement to contain it. When nodes overflow, i.e. the number of points they contain exceeds a pre-defined threshold, the node is "split", creating two child nodes into which points are assigned. R-Trees can be used for spatio-temporal data by assigning the time to one dimension.

*2.3.1. R-Tree Node Splitting.* Guttman originally proposed 3 bi-partition algorithms for the R-Tree:

(1) The linear / sort-based algorithm sorts points along a dimension and then splits the resulting list in half.
(2) The quadratic / seed-based algorithm finds the two most distant points (seeds) in a node then associates the other points to the seed which is the closest.
(3) The exponential / exhaustive split algorithm explores all the possible combinations and chooses the one with the lowest coverage or overlap.

Node splitting is a vast subject of research and a number of algorithms have been proposed to extend this approach.

For instance, `Double-Sorting` [Korotkov 2012] offers the query performance of Guttman's quadratic split at the cost of the linear split. It searches for points whose coordinates can divide the node with minimal overlap. This algorithm based on two sortings allegedly offers better splitting on complicated datasets.

The `Packed R-Tree` [Roussopoulos and Leifker 1985] extends Guttman's linear sorting to trees with more than 2 children, sorting $N$ points along one dimension, then packing them into $\lfloor \frac{N}{n} \rfloor$ consecutive groups of $n$ points. This process is carried out recursively until the groups have the desired number of points or desired number of subgroups.

The `cR-tree` [Brakatsoulas et al. 2002] considers that node-splitting is a clustering problem, which can extend further than to 2 children. The authors use MacQueen's popular k-means [MacQueen 1967] algorithm in order to split nodes into multiple children.

*2.3.2. R-Tree Variants.* `R`$^+$`-Trees` [Sellis et al. 1987] are variants of the R-Tree in which sibling nodes have no overlap. Indeed, the node splitting algorithm allows rectangles to be split, then inserted into several nodes.

`R`$^*$`-Trees` [Beckmann et al. 1990] attempt to minimize overlap and coverage by reinserting points from overflowing nodes into the tree, thus reducing the effect of the initial order of the points on the tree's structure. Node split is deferred, in order to ensure that the resulting nodes' fill is higher. Finally, R*-Trees use a topological split that minimizes overlap.

The `Hilbert R-Tree` [Kamel and Faloutsos 1994] is an R-Tree variant in which points are ordered and grouped according to their value along the Hilbert curve or other space-filling curves. The fractal properties of these curves tend to group close points together, thus minimizing the overlap and area of nodes.

`Historical R-Trees` (HR-Trees) [Nascimento and Silva 1998] are a modification of R-Trees to support versioning, by building an R-Tree for each timestamp and sharing common nodes with links between trees. This is efficient when there are few modifications between a small number of versions.

The `MV3R-Tree` [Tao and Papadias 2001] combines a Historical R-Tree – for the infrequent state changes and a 3D R-Tree – for the shorter lived states, to benefit from the properties of each tree on the type of data they are better suited for.

*2.3.3. External Memory R-Trees.* The `Small-Tree-Large-Tree` (STLT) [Chen et al. 1998] method makes it possible to bulk load points into a tree while reducing the time during which it is unavailable for queries. It proceeds by creating a new R-Tree on the side – the Small-Tree – into which the points are loaded. Then, this tree is inserted into the optimal position of the main tree – the Large-Tree. This approach works particularly well for skewed data.

The `Generalized R-Tree Bulk-Insertion Strategy` (GBI) [Choubey et al. 1999] generalizes the STLT approach for less skewed data. The data to bulk load is split into clusters and outliers with a variation of the K-means algorithm. Each cluster is bulk loaded separately with the STLT method, while the outliers are inserted as single points into the tree.

The `Buffer R-Tree` [Biveinis et al. 2007] takes avantage of the system's main memory to reduce external memory I/O, by delaying insertions or deletions to the external memory structure until a certain number of operations can be bulk executed in a more efficient fashion.

In this paper, we propose a scalable data structure, which has performance gains compared to previous implementations, is well suited to parallel systems, and offers much improved query times.

## 3. LIMITATIONS OF THE STATE HISTORY TREE

In this section, we briefly present the implementation of the State History Tree (SHT) [Montplaisir et al. 2013], a data structure designed for state storage on external memory, and detail the issues that it encounters when dealing with highly parallel traces.

### 3.1. Structure of the State History Tree (SHT)

The State History Tree (SHT) [Montplaisir-Goncalves et al. 2013] is suited for storage of stateful information that is computed while reading through the trace. It is used for tracking the states of various state machines over the duration of a trace. For example, it is possible to know the status of a process at any time in the trace, when analysing a Linux kernel trace. Trace events produce the transitions in the state machine, and stateful analysis computes the states between transitions, before storing them in the SHT. During the trace analysis, events are processed in chronological order, therefore we track unclosed states with their start times. Every time an event changes a state, we write the ending state interval to external memory and update the current state value and start time.

The stored data takes the form of intervals, which consist of an attribute key, a start time, an end time and a value: $< key, time_{start}, time_{end}, value >$. The start and end times are specified with a nanosecond granularity. The attribute key is a unique identifier for the object whose state we are tracking. The value is the payload of the interval which can be a null, a boolean, an integer, a long or a character string. For each attribute, there are contiguous intervals from the beginning until the end of the trace.

The SHT is composed of nodes created as the tree is built. A node is defined by a unique sequence number, a start time and an end time. They have a header which contains the sequence numbers of their parent as well as the sequence numbers and start times of their children, for search. Each node is mapped to blocks on storage media, and can therefore store a limited number of intervals. As the serialized size of intervals can vary on the type of value they carry, this number is not fixed. The maximum number of children in a SHT node is limited – usually to 50 – and its capacity is limited to $64kB$. The unique sequence numbers of the nodes represent the position of their block in the history tree file.

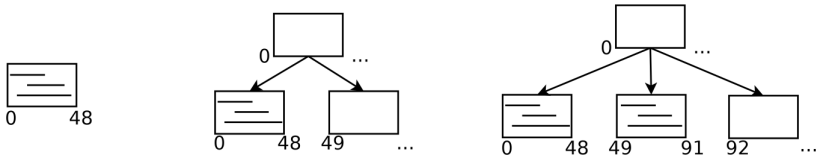



Fig. 2: Build steps of the State History Tree using an incremental process [Montplaisir et al. 2013]

The tree is built in a single pass upon performing the state analysis of a trace. As events in the trace are in chronological order, the resulting intervals are generated and inserted with increasing end times. The tree's construction starts from a single node, that has the same start time as the trace. The rightmost branch of the tree is kept in main memory so that intervals can be efficiently inserted into them while the rest of the nodes are kept serialized in external memory until it is necessary to read from them. As a node's time range must be included in its parent, start times of the nodes in the rightmost branch increase from the leaf to the root node as shown in Figure 2. Therefore intervals are inserted into the deepest node with a start time smaller than that of the interval. Once a node $node_{full}$ has reached the maximum number of children or interval capacity, all the rightmost children from itself to the leaf have their end times set to that of the last interval which was inserted $time_{last}$ and are written to external memory. If the node $node_{full}$ has a parent, the rightmost branch is rebuilt with nodes starting at $time_{last} + 1$ to ensure that the tree is balanced. If $node_{full}$ was the root node, a new node, starting at the start time of the trace, becomes the new root node.

### 3.2. Shortcomings on analysis of large systems

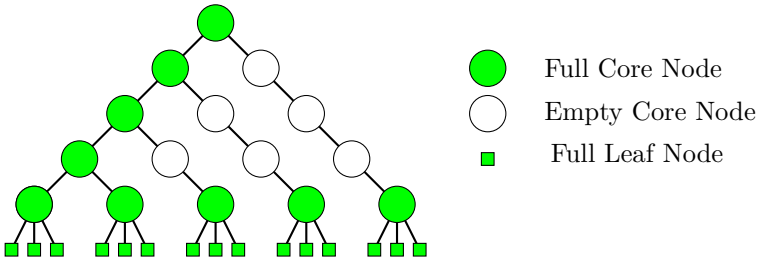



Fig. 3: Schematization of a State History Tree with many attributes

We encountered traces for systems with a large number (30000) of threads, which had a reasonable size (less than $100Mb$), yet created a large SHT file with inefficient storage usage (around 9% of the storage space reserved by the many nodes is actually used to store interval data). Moreover, these traces brought our visualization software to a halt, leading to further investigation.

Since we store intervals from the tree's start time for every attribute (such as threads), the SHT has many intervals that begin at the SHT's start time. The SHT's design imposes that intervals that begin at the SHT's start time fit in the left most node. Therefore, a trace for a many-threaded program leads to a very deep tree compared to a well balanced, high fan-out tree. Indeed, the depth of the tree increases when we try to insert an interval beginning at the tree's start time and it cannot fit into the deeper non filled nodes, as its start time is earlier than theirs. Therefore, it can only fit in the root node and, when the

root node is full, another depth level is added. In the case of our trace with many threads, the tree was 206 nodes deep, whereas an efficiently loaded tree for the same data should have been 5 nodes deep.

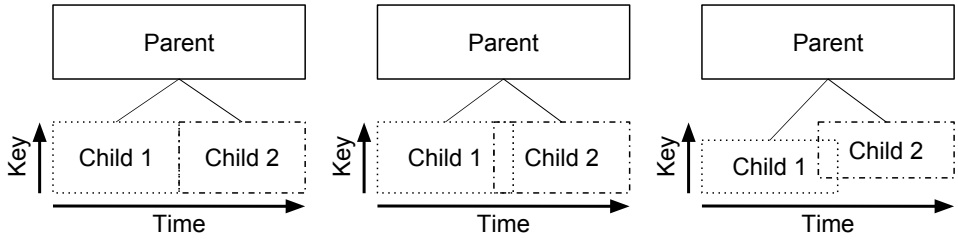
We have a very deep tree but most intervals fit in leaf nodes, therefore the branches are mostly empty from the leftmost node to the leaves. As the SHT nodes are mapped to external memory blocks, many empty nodes means a very low storage usage rate. As queries on an SHT search down a branch for which the nodes cover the queried time, and most intervals are stored in the leaf nodes, the average query takes a long time.

Therefore, we can compute the depth $d$ of a SHT with many attributes, as $d = \frac{A}{n}$, with $A$ the number of attributes and $n$ the average number of attributes that fit per node. This differs from the depth of a packed, balanced tree $d = \lceil log_c(\frac{N}{n}) \rceil$, of degree $c$ with $\frac{N}{n}$ nodes, $N$ being the number of intervals in the tree.

Likewise the number of nodes of a *comb* tree, such as the one in Figure 3 will be $\left(\frac{A}{n} - 1\right)\left(\frac{\frac{A}{n}-1}{2} + c\right)$ instead of $\frac{N}{n}$.

Because the tree is very deep and the State System stores its "in progress" branch in memory, the SHT construction may even crash with a JVM OutOfMemoryError on the deepest trees.

## 4. THE OVERLAPPING STATE HISTORY TREE




(a) Representation of consecutive nodes in SHT

(b) Representation of overlapping nodes in oSHT

(c) Representation of enhanced nodes in eSHT

Fig. 4: Comparison of the relations between sibling nodes of SHT (left), oSHT (middle) and eSHT (right)

### 4.1. Overlapping SHT structure

The overlapping SHT structure has been introduced as the enhanced State History Tree (eSHT) [Prieur-Drevon et al. 2016]. This structure fixes the original SHT's tendency to degenerate into combs in cases with many attributes.

As explained above, imposing that sibling nodes' time ranges be consecutive causes the tree to degenerate into combs. The consecutive (non overlap) constraint is shown in Figure 4(a). We do away with this constraint and use the first inserted interval's start time as the new node's start time. The removal of this constraint ensures that intervals inserted in the future fit into the leaf nodes, which in turn prevents the tree's depth from degenerating. However, the overlap shown in Figure 4(b) requires modifications to the data structure and algorithms.

Because sibling nodes overlap, we modify the query algorithm – described in Algorithm 1 – to query on sub-trees instead of branches. The different search algorithms are represented respectively in Figure 5(b) and 5(a). In order to query the correct sub-tree, the children's

end times are added to their parent node's header, alongside their start times. A node's time range must still be included in its parents time range.

There are a number of benefits to allowing the nodes to overlap. We can now fit a large number of attributes that cover the same time ranges, which is typical for highly parallel trace analysis, without degenerating into a list. Therefore, the tree should be shallower, so queries may be shorter and the build would use less memory. There should also be far less empty nodes, so better use of storage space would be made, reducing the number of writes when building the tree (and consequently, build times).

### 4.2. Enhanced State History Tree

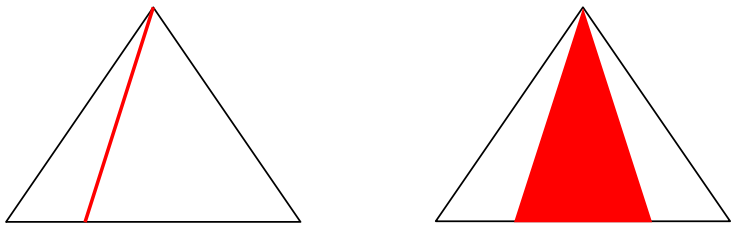
We now consider the data we are handling as multidimensional (time and key), so we add key bounds to the header. These describe the minimum and maximum keys for the intervals stored in the node, as shown in Figure 4(c). The core nodes' headers also store their children's bounds, to help narrow down the number of nodes searched during a query. As was the case with the time bounds, a nodes key bounds must also be included in its parents bounds. For example, if a core node's header says that one of it's sub-trees has bounds $[t_{min}, t_{max}]$ for the time and bounds $[k_{min}, k_{max}]$ for the keys, there is no point in searching it for a key $k_s$ such that $k_s < t_{min}$ or $k_s > t_{max}$. We call *enhanced State History Tree* (eSHT) the overlapping State History Tree with added key bounds.

### 4.3. Search algorithms

The SHT can be queried for the state of one or all attributes at a time $t$.

— A single query for key $k$ at time $t$ returns the interval for key $k$ that overlaps $t$. The single queries search down the branch of the tree that overlaps $t$ until they find the interval with the correct key and time range.
— A full query at time $t$ returns all the intervals (one per key) that overlap $t$. Full queries search the entire branch that overlaps $t$, and add all the intervals that overlap $t$ – one per attribute – to a list

Concurrent accesses are handled by using a shared-exclusive lock for the current states and one shared-exclusive lock per node.



(a) Representation of a branch search as used by SHT

(b) Representation of a sub-tree search as used by eSHT

Fig. 5: Comparison of tree search between SHT (left) and eSHT (right)

Unlike for SHT and slog2, queries on eSHTs cover a sub-tree, and not just a branch in the tree, since there are potentially several children nodes that contain the relevant time. Therefore, each core node contains an index on the start and end times of each of its children

to determine which sub-tree to explore. The method **node.getChildren(t)** produces a list of **node**'s children which contain time $t$ in Algorithm 1. The **node.getInterval(k, t)** method retrieves the interval intersecting $t$ with key $k$ when it exists in **node**.

**ALGORITHM 1:** Single State Query

```
function singleQuery(key, time)
    interval ← null
    /* rootNode is the tree's root node.                    */
    queue ← List(rootNode)
    while interval = null do
        node ← queue.pop()
        if node.type() = CoreNode then
            queue.addAll(node.getChildren(key, time))
        end
        interval ← node.getInterval(key, time)
    end
    return interval
end
```

### 4.4. Comparison of query bounds with SHT

The State History Tree aims for query performance, so we consider that the number of nodes searched per query is a good measure of the data structure's query efficiency. We will theoretically compare the number of nodes searched for queries on our enhanced State History Tree to the original one.

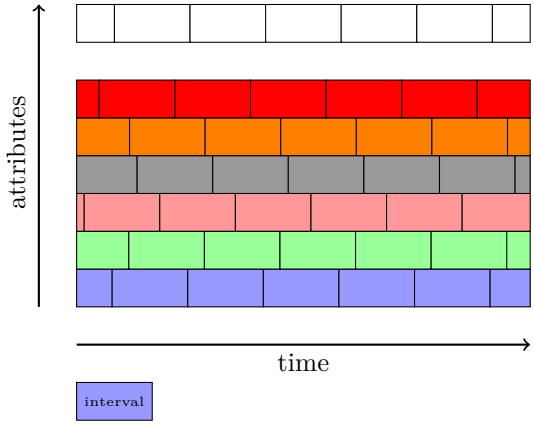



Fig. 6: Schematization of the intervals in a Tree

The theoretical intervals used for comparison are represented in Figure 6. That tree is of duration $T$ and contains $A$ different attributes. Each attribute is split into $I$ equal intervals through the trace. Intervals from different attributes are offset by $\frac{T}{AI}$. The order of the Attributes is shuffled. Each node can store up to $n$ attributes and have up to $c$ children.

On the SHT, queries search down a branch, as explained in 4.3, therefore the upper bound is the tree's depth. With our comparison trace set, with its large number $A$ of attributes, we will reach the comb situation. The query bound for SHT $Q_{SHT}$ can thus be formulated in this way:

$$Q_{SHT} = \left\lceil \frac{A}{n} \right\rceil$$

As for the eSHT, we need to determine the number of nodes which overlap the queried time. The query bound $Q_{eSHT}$ for eSHT can then be expressed as the following:

$$Q_{eSHT} \leq h + \left( \frac{A+n}{n+1} \right) \times \left( \frac{1-c^{-h}}{1-c^{-1}} \right)$$

With $h$ being the depth of the eSHT and $c$ the maximum number of children per node. Considering that, in our example, all the data is in the leaf nodes, we can use the standard formula [Becker et al. 1996] to compute the tree's height $h$:

$$h = log_c \left( \frac{AI}{n} \right)$$

We approximate $Q_{eSHT}$ by computing the required number of nodes to fit $A$ intervals. Due to the algorithm used to build eSHT and our worst case theoretical trace, all the intervals reside in leaf nodes. As intervals are inserted by increasing end times, the node duration $D$ is:

$$D = \frac{T}{I} + n \times \frac{T}{AI}$$

Where $\frac{T}{I}$ is the interval duration, for the average interval, i.e. neglecting border effects for the first and last intervals of each attribute. We call $\Theta$ the number of nodes in the tree which overlap a time $t$. It can be computed as the ratio of the node duration $D$ over node offset $\Delta t$:

$$\Theta = \frac{D}{\Delta t}$$

Which is then:

$$\Theta = \frac{\frac{T}{I} + n \times \frac{T}{AI}}{(n+1) \times \frac{T}{AI}}$$

And can be reduced to:

$$\Theta = \frac{n+A}{n+1}$$

However, we also have to consider the core nodes which have to be searched to reach the leaves from the root node. Knowing that each core node can reference up to $c$ children, we deduce the following relation:

$$Q_{eSHT} = \sum_{i=0}^{h} \left\lceil \frac{\Theta}{c^i} \right\rceil$$

And then develop the upper bound for $Q_{eSHT}$:

$$Q_{eSHT} \leq \sum_{i=0}^{h} \left( \frac{\Theta}{c^i} + 1 \right)$$

That can then be reduced to the following:

$$Q_{eSHT} \leq \Theta \times \frac{1 - c^{-h}}{1 - c^{-1}} + h$$

While this is slightly larger than the query on a SHT, the average query size on an eSHT is half that of the upper bound, as the intervals are uniformly spread over the possible DFS or BFS search path.

Meanwhile, on the SHT, most branches are empty, as seen in Figure 3. We can compute the average depth of intervals in the tree, knowing that intervals are either in the left most nodes, in the deepest core nodes, or in the leaf nodes. If we consider:

— $H$ as the height of the tree
— $N_{\text{leaf}} = c(H-1)$ as the number of **leaf** nodes, of depth $H$
— $N_{\text{core}} = (H-1)$ as the number of **core** nodes, of depth $H-1$
— $N_{\text{left}} = (H-2)$ as the number of **left** nodes, with increasing depths from 0 to $H-2$.

We can express the average depth of nodes containing intervals as the following:

$$d_{\text{avg}} = \frac{\sum d}{\sum N}$$

Which can be developed as:

$$d_{\text{avg}} = \frac{\sum_{i \in \text{leaf}} d + \sum_{i \in \text{core}} d + \sum_{i \in \text{left}} d}{N_{\text{leaf}} + N_{\text{core}} + N_{\text{left}}}$$

And thus:

$$d_{\text{avg}} = \frac{\sum_{i=0}^{H-2} i + (H-1)^2 + c(H-1)H}{H - 2 + H - 1 + c(H-1)}$$

As we know that $H \gg 1$, the equation can finally be reduced to:

$$d_{\text{avg}} \simeq H$$

Therefore, the average eSHT query on traces with many attributes is close to twice as fast as the average SHT query.

### 4.5. Query scalability limitations

In early results, we modeled the behavior of the SHT and the eSHT and found the number of nodes that needed to be searched for queries. We found an upper bound of $\frac{A}{n}$ nodes, with $A$ being the number of attributes and $n$ the average number of intervals per node. This upper bound was similar for both trees and the average single query was only twice as fast in the case of the eSHT, as can be seen in Figure 7.

To try and understand why the eSHT does not speed up single queries more than twofold, we look at the key range covered by the key bounds discussed in Section 4.2. We define the key range by the difference between each node's minimum and maximum key.

The key range histogram in Figure 8 shows that the key range distribution is sub-optimal. Indeed, if the intervals had been arranged efficiently in the tree, the range would have been minimized for deeper nodes and only shallower nodes would have contained the entire range.

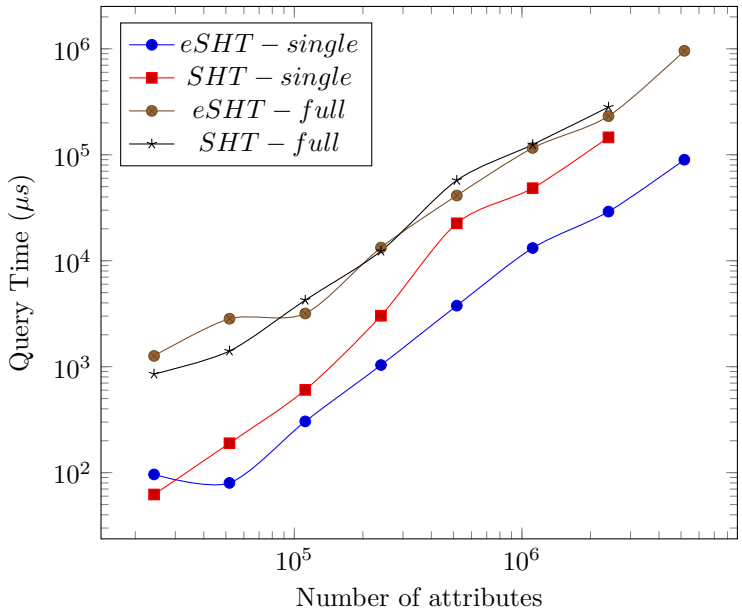



Fig. 7: Comparison of SHT and eSHT query times for traces with many attributes

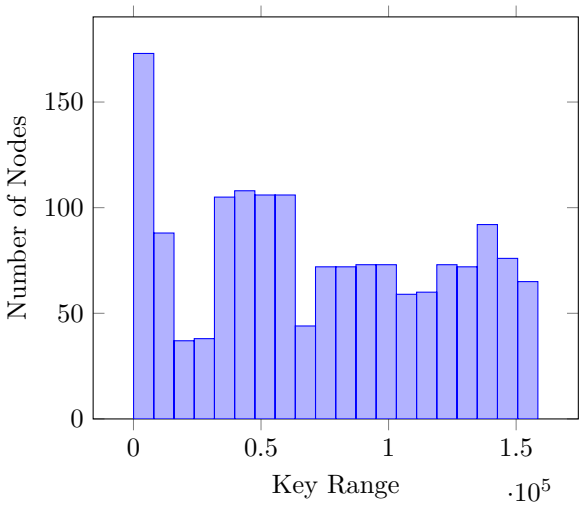



Fig. 8: Key range histogram for 10k thread trace using eSHT

## 5. R-SHT MODEL, STRUCTURE AND ALGORITHMS

In this section, we propose a modification inspired by R-Trees. To prove its relevance and performance gains, we develop a model to describe the data structure's behavior and quantify the expected performance gains.

### 5.1. R-Tree qualities for the SHT

In early results, we suggested considering the SHT as a two dimensional data structure, with the time and attribute dimensions. However, indexing the key values provided only minor

gains on search performance, compared to more optimized R-Trees in particular. Queries on R+-Tree structures [Sellis et al. 1987] search through as many nodes as the tree is deep. This was due to a sub-optimal organization of intervals in the nodes : the average node covered a wide range of attributes. As the MBR overlap remained high (Figure 9), single queries were only slightly narrowed down.

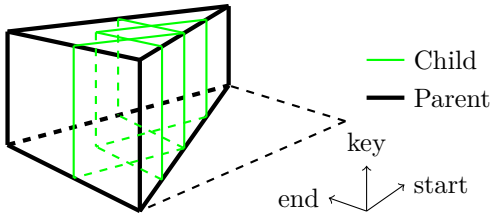



Fig. 9: Default oSHT Split

To improve the query performance we want to provide the properties of an optimal R-Tree to the SHT. However, [Montplaisir-Goncalves et al. 2013] has shown that Guttman R-Trees had worse insertion performance than the SHT, due to chronic re-balancing and was ill suited to large traces. Indeed, frequent re-balancing required in memory processing for performance reasons, while large traces would not fit in main memory.

Therefore we propose a R/SHT hybrid using Small-Tree-Large-Tree and Bulk Loading techniques. We consider using the eSHT structure for the upper nodes and buffering shorter intervals before inserting them into R-Trees in the lower level nodes.

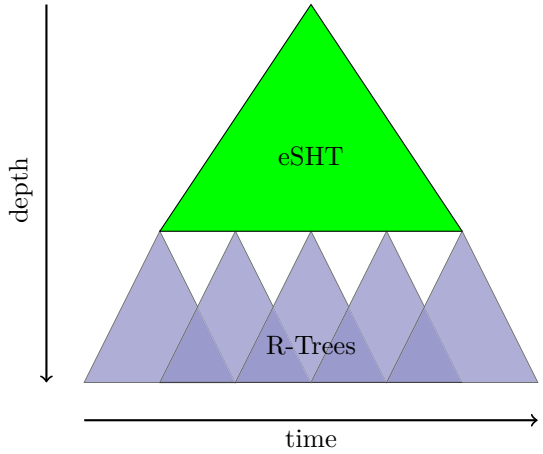



Fig. 10: R-SHT hybrid structure

### 5.2. Build Algorithm

The build algorithm works similarly to the eSHT, with intervals with earlier start times inserted into shallower nodes. However, intervals starting later than the deepest SHT node go into a temporary buffer. This buffer has the capacity of an eSHT of the same depth, so that the intervals can be mapped to eSHT nodes once built.

Algorithm 2 presents the algorithm that we use to move intervals from the R-Buffer into the eSHT when the buffer is full. This algorithm works top down, from the R-Buffer's root

**ALGORITHM 2:** Top Down Buffer to R-SHT copy algorithm

```
function rBuild(intervals, node, height)
    if node = LEAF then
        node.insert(intervals)
    else
        node.insert(intervals.outliers())
        for cluster ∈ intervals.cluster() do
            rBuild(cluster, new child(), height − 1)
        end
    end
end
```

node level, to the leaf node level, and allows for additional flexibility by choosing the most relevant **outliers** and **cluster** methods.

### 5.3. Clustering Algorithm

As explained in [Choubey et al. 1999], the choice of the **outliers** and **clustering** functions in Algorithm 2 plays an important role in determining in which node the intervals will be inserted.

We consider the intervals from Figure 6. There are at most $\frac{n}{\delta q}$ intervals per attribute in a node, with $\delta q$ the node's attribute range and $n$ the number of intervals per node. The time span $\delta t$ of node's MBR is the sum of complete intervals' durations and offsets between attributes:

$$\delta t = \frac{n}{\delta q}\frac{T}{I} + \delta q \times \frac{T}{AI}$$

With $T$ the total trace duration, $A$ the total number of attributes and $I$ the number of intervals per attribute.

We want to minimize the nodes' area:

$$\min(\delta q \times \delta t) = \min\left(\left(n + \frac{\delta q^2}{A}\right) \times \frac{T}{I}\right)$$

Which is equivalent to minimizing $\delta q$.

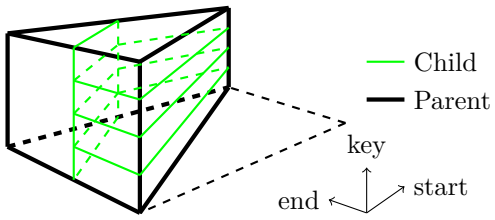



Fig. 11: Splitting the R-Tree Buffer along the key dimension

Therefore, the splitting which minimizes the nodes' area minimizes the range along the query dimension. We move longest intervals into the parent node as outliers, since they are not accounted for by the model and naturally go into the shallower nodes in SHTs and following implementations. The remaining intervals are sorted by their key, and consecutive sub-ranges are mapped to sub-trees.

Figure 11 shows the split performed by the key sort algorithm, with the red plane separating the longest intervals, which go into the parent node, from the shorter intervals, which go into the children. Finally, the green planes show the split along the key dimension.

### 5.4. 2D Queries

When we need to extract data from the tree for multiple timestamps and multiple attributes at those timestamps, the current solution was to use single or full queries. Such approaches respectively mean multiple requests or a request that returns unneeded intervals. We thus introduce 2D queries to make extracting data from the tree faster. These queries take as arguments a list of keys and either a time range or a list of timestamps, depending on the requirement. They return a map, where each queried key's associated value is the ordered lists of intervals matching the query:

```
Map(key -> intervalList) 2Dquery(keyList, timeList)
Map(key -> intervalList) 2Dquery(keyList, start, end)
```

The query is executed recursively from the root node. The time and key lists are narrowed down from the parent's to the child's bounds by using binary searches. Sub-trees which bounds do not contain any element from the time and key lists are not searched. The outcome of these queries is to execute a single search of the tree that returns all of the required intervals and only the required intervals.

## 6. PERFORMANCE

In this section we look at how much faster we can make the single queries, full queries and 2D queries with the R-SHT compared to a typical eSHT or SHT. We use the intervals model from Figure 6, stored in an R-SHT with a R-Tree section of depth $r$, up to $c$ children per node and $n$ intervals per node.

For all the models on R-SHT, determining the number of nodes searched will consist in computing how many nodes are searched in the upper / eSHT levels and how many nodes are searched in each of the R-Trees of the structure:

$$S_{R-SHT} = S_e + n_R \times S_R$$

With $S_e$, the number of nodes searched in the upper / eSHT levels of the tree, $n_R$ the number of R-Trees searched and finally $S_R$, the number of nodes searched per R-Tree.

### 6.1. Single queries

The number of R-Trees that are queried during a single query is the number of R-Trees that overlap the queried time. The R overlap for any time is the R-Tree's duration $D_R$ over the R offset $\delta t_R$:

$$n_R = \frac{D_R}{\delta t_R}$$

With the R-Tree's duration being the sum of all the intervals' two-by-two offsets and a full interval duration:

$$D_R = c^{r-1} n \frac{T}{AI} + \frac{T}{I}$$

And the R offset being the sum of all the intervals' two-by-two offsets:

$$\delta t_R = c^{r-1} n \frac{T}{AI}$$

Therefore $n_R$ is:

$$n_R = \frac{c^{r-1}n + A}{c^{r-1}n}$$

The number of nodes searched in the eSHT levels $S_e$ is also the same, whichever R-Tree clustering algorithm is chosen. The eSHT levels are $log_c(\frac{AI}{n}) - r$ nodes deep. We compute the overlap for a specific time for every level considering each node's duration $D_d$ to be that of it's sub-tree.

$$D_d = \min\left(c^{h-d}n\frac{T}{AI} + \frac{T}{I}, T\right)$$

And the level's offset $\delta t_d$ is the sum of the sub-tree's intervals' offsets:

$$\delta t_d = c^{h-d}n\frac{T}{AI}$$

Therefore, the number of nodes searched in the eSHT levels is the total number of nodes that overlap $t$ in the eSHT levels, or the sum of overlaps per level:

$$S_e = \sum_{d=0}^{h-r} \frac{D_d}{\delta t_d}$$

Which can be developed as:

$$S_e = 1 + \sum_{d=1}^{h-r} \frac{c^{h-d}n + A}{c^{h-d}n}$$

That can be computed as:

$$S_e = h - r + \frac{A}{n}\frac{c(c^{-r} - c^{-h})}{c-1}$$

And is approximately equal to:

$$S_e \simeq h - r + c^{-r}\frac{A}{n}$$

For single queries on an R-SHT built with the key-sort algorithm, the intervals have been sorted by their key, so we know into which R-Tree branch to search, as there are fewer intervals per attribute in the R-Tree than can fit into a node.

$$S_R = \left\lfloor r + \frac{c^{r-1}}{A} \right\rfloor$$

Therefore, the total number of nodes that must be searched in the full tree is:

$$S_{R-SHT} = h - r + c^{-r-1}\frac{A}{n} + \frac{c^{r-1}n + A}{c^{r-1}n}\left\lfloor r + \frac{c^{r-1}}{A} \right\rfloor$$

Which approximates to:

$$S_{R-SHT} \simeq h + \frac{A(1+r)}{c^{r-1}n}$$

This is faster than on the SHT for which the number of nodes to be searched is $\frac{A}{n}$, as the R-SHT's height $h$ is orders of magnitude smaller than $\frac{A}{n}$ and $c >> r$

### 6.2. 2D Query

There are too many argument combinations to compute the average complexity of the 2D queries introduced in section 5.4. We can however try to determine the number of nodes searched in the worst case scenario.

The main advantage of the 2D query is that even for long lists of queried keys and times, it will never search a node more than once. Therefore, its upper bound is the number of nodes in the tree:

$$bound = \frac{AI}{n}$$

When the key and time lists are very sparse, e.g. with respectively $k$ and $t$ items, the worst case is similar to returning each interval with a single query $SQ$:

$$sparse\_bound = k \times t \times SQ$$

We modeled the 2D queries' performance to be faster than that of the equivalent single or full queries to extract the equivalent information.

### 6.3. Full queries

Full queries return the states of all the intervals intersecting a specified timestamp $t$, therefore the $S_{eSHT}$ and $n_R$ values for these queries are the same as for single queries.

However, on the key-sort R-SHT, the $S_R$ value becomes equal to the number of nodes in the R-Tree levels. Indeed, by sorting intervals by keys, we end up spreading intervals overlapping $t$ over the entire sub-tree.

$$S_R = c^r$$

Therefore, we see that full queries on an R-SHT will be noticeably longer than on a SHT:

$$S_{R-SHT} = h - r + c^{-r}\frac{A}{n} + \frac{c^{r-1}n + A}{c^{r-1}n} \times c^{r-1}$$

Which approximates to:

$$S_{R-SHT} \simeq h - r + c^{r-1} + \frac{A}{n}$$

R-SHT does not aim to speed up the full queries, as they are an inefficient implementation and can easily be replaced by single or 2D queries. The previous model shows that they are indeed slower than on the SHT, which searches $\frac{A}{n}$ nodes.

### 6.4. Impact of the buffer size

We use the height of the R-Buffer as the degree to which the organization of the data structure will be optimized. The higher said tree, the faster the queries, at the cost of longer optimization and tree build process.

However the required depth of the R-Buffer is linked to the key range, which is rarely known in advance. We deal with this situation by increasing the height of the buffer during

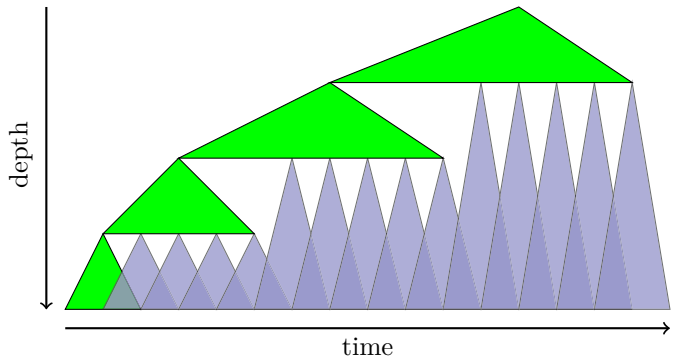



Fig. 12: Increasing the depth of the R-Tree buffer during construction

the tree construction, when the maximum key exceeds a certain threshold. As can be seen in Figure 12, the height can be increased once the current R-Tree's parent is full, and a new parent needs to be initialized. This new parent will be the root node of the new, deeper R-Tree.

We can define the threshold such that consistent query performance is maintained, whichever the attribute range, once we know the theoretical query performance relative to the attribute range $A$ and R-Tree buffer's height $r$.

Knowing that the single queries complexity is $S_{eSHT} \simeq h - r + \lceil c^{-r} \frac{A}{n} \rceil$, we must find $r$ such that we search the same number of nodes for all values of $A$. We want that:

$$S_{R-SHT}(A) \leq S_{SHT}(A_{arb})$$

With $A_{arb}$ an arbitrary number of attributes chosen to reflect satisfactory SHT performance. That can be developed:

$$h + c^{-r-1} \frac{A}{n} \leq \frac{A_{arb}}{n} + r$$

From which we compute:

$$r = \left\lceil log_c \left( \frac{A}{n} \right) \right\rceil$$

The benefits of increasing the depth of the buffer only when needed are that we no longer need to determine the depth in advance. Furthermore, trees with fewer attributes, which require less optimization, can be built faster than if the depth was hard-coded or computed otherwise.

## 7. RESULTS

In this section we look into the performance results of the SHT, eSHT and R-SHT on trace analysis workloads.

### 7.1. Test Environment

All experiments were conducted on an Intel Core i7 3770 @ 3.4GHz with 16 GB RAM, a Samsung 850 PRO-Series 512GB Solid State Drive, using Eclipse version 4.5.1 and OpenJDK version 1.8.0_66. The trace files were generated using LTTng version 2.7.0 and the Debian Linux kernel version 4.5.0-2-amd64. We set the kernel's maximum PID value to $2^{22}$ up from its default value of $2^{15}$, so that each process will have a unique PID. The trace files contain the detailed execution trace at kernel level, including all the system calls, scheduling events and interrupts. For the following benchmarks, we traced a burn program, which creates many parallel threads, leading to many attributes in the State System.

We use `Trace Compass`' Kernel Analysis to perform our benchmarks and generate the History Trees. This analysis reads all the events from the trace and tracks various attributes by storing them in the SHT. In particular, there are several attributes stored for each thread and CPU of the analyzed system. This means that the bigger the trace is, and the more activity there was on the system during the trace, the bigger the generated SHT will be. This analysis will thus allow us to perform a thorough comparison between the original SHT (labelled `SHT` in the following experiments), the enhanced SHT (labelled `eSHT`) and R-SHT metrics. We will compare R-SHTs of heights from 2 to 4 (labelled `R2`, `R3` and `R4`), as well as the variable height R-SHT (labelled `vR`), with maximum height of 4.

For our scalability benchmarks, we generate the traces in table I, to demonstrate the scalability of our solution to traces with many threads for example.

Table I: Scalability trace set specifications

| # threads | # events | size |
|---:|---:|---:|
| 160 | 7266 | 600K |
| 336 | 13133 | 772K |
| 736 | 19979 | 1004K |
| 1600 | 27722 | 1.3M |
| 3440 | 57721 | 2.3M |
| 7424 | 122552 | 4.4M |
| 16000 | 273666 | 8.9M |
| 34464 | 585565 | 19M |
| 74256 | 1263257 | 40M |
| 160000 | 2721160 | 85M |
| 344704 | 5917942 | 184M |
| 742640 | 12619069 | 390M |
| 1600000 | 27159383 | 839M |
| 3447088 | 58640566 | 1.8G |

### 7.2. Case Study: Control Flow View

We look into how the R-SHT can be exploited to make Trace Compass faster. In particular, Time Graph views, such as the one in Figure 13, which show the states of a list of attributes through time, require either many queries to obtain the required intervals or more complex ones that cover a larger time range.

Table II: Comparison of the History Trees for the `many-threads` trace

| | Build (ms) | | | Depth | Size (MiB) |
|---:|---:|---:|---:|---:|---:|
| **SHT** | 1 925 | ± | 353.1 | 45 | 86.31 |
| **oSHT** | 1 812 | ± | 332.9 | 3 | 21.87 |
| **eSHT** | 1 762 | ± | 345.4 | 3 | 21.44 |
| **R2** | 6 604 | ± | 727.9 | 3 | 20.87 |
| **R3** | 8 926 | ± | 1 485 | 3 | 20.81 |
| **vR** | 6 546 | ± | 387.4 | 3 | 21.00 |


Fig. 13: The Control Flow View, a Time Graph View in Trace Compass

We consider the `many-threads` trace from the Trace Compass test package. It is a 7.73 MiB trace from LTTng 2.8 with 240644 events. The resulting Kernel Analysis generates 741492 intervals, for 50598 attributes with a raw size of 18.62 MiB. We compare the build times, depth and file size for each implementation of the History Tree in Table II. As the R-SHTs are 3 nodes deep, we do not consider R4 as it is indistinguishable from R3. The original SHT stands out by its excessive depth and file size.

We look into two metrics related to displaying the Control Flow View: `PsTree` is the time to build the process tree on the left hand column of the Control Flow View, which requires each thread's *PPID* and *Exec_name*. `Zoom` is the total time for an automated sequence of scrolling and zooming the View, including the query to the State History Tree and the post processing of the returned states. The sequence begins by filling in the states for a completely zoomed out view (i.e. the trace timeline fills the entire width of the view), then geometrically zooms in ten times, and horizontally scrolls from the beginning to the end of the trace.

For each of those metrics, we compare three querying strategies: **Full** is the currently implemented strategy. It does a full query for every horizontal pixel. Between each full query, the returned values are used to build the `PsTree` and render the threads' statuses. **Single** does single queries for each thread's *PPID* and *Exec_name* for the `PsTree` and does single queries for the visible processes when zooming. **2D** does a query which returns all the *PPID* and *Exec_name* for the threads in one pass and another query per zoom to return a downsampled set of the visible processes' statuses.

The current implementation, which we will use as a baseline, is the SHT with full queries. The 2D query is able to extract all the required information to populate either the PsTree or the States in a single pass of the tree. Because the key dimension is defined from the eSHT version onwards, we cannot provide such results for SHT and oSHT.

As expected, the full query is the most efficient with the SHT, as that is the implementation for which said query has to search through the least number of nodes.

However, the PsTree phase requires information from every thread. These keys are spread out over the attribute range, and concern relatively few intervals. Therefore, it is better suited to the 2D queries, as Table III shows. Indeed the 2D query is bounded by the number of nodes in the tree. The full query implementation would search entire SHT branches or the full R3 tree for every horizontal pixel.

Table III: Gains over SHT for displaying data in a Control Flow View. (mean ± standard deviation) on 10 executions.

| Query Type | PsTree (ms) | | | Zoom (ms) | | |
|---|---|---|---|---|---|---|
| **SHT** | | | | | | |
| *full* | 5 360 | ± | 645.9 | 29 880 | ± | 547.6 |
| *single* | 21 210 | ± | 860.4 | 15 300 | ± | 314.1 |
| **oSHT** | | | | | | |
| *full* | 6 536 | ± | 849.4 | 26 860 | ± | 715.8 |
| *single* | 19 500 | ± | 588.1 | 15 000 | ± | 143.3 |
| **eSHT** | | | | | | |
| *full* | 6 168 | ± | 396.0 | 30 310 | ± | 1 432 |
| *single* | 940.6 | ± | 80.41 | 6 894 | ± | 122.6 |
| *2D* | 226.8 | ± | 25.61 | 4 408 | ± | 174.9 |
| **R2** | | | | | | |
| *full* | 19 220 | ± | 4 143 | 42 310 | ± | 487.3 |
| *single* | 1 882 | ± | 43.35 | 7 861 | ± | 123.1 |
| *2D* | 136.3 | ± | 28.57 | 3 950 | ± | 183.1 |
| **R3** | | | | | | |
| *full* | 65 590 | ± | 11 730 | 267 000 | ± | 9 780 |
| *single* | 914.9 | ± | 48.34 | 6 995 | ± | 168.9 |
| *2D* | 118.4 | ± | 53.09 | 4 422 | ± | 870.4 |
| **vR** | | | | | | |
| *full* | 17 100 | ± | 7 261 | 40 970 | ± | 2 878 |
| *single* | 1 969 | ± | 55.73 | 8 183 | ± | 261.0 |
| *2D* | 135.7 | ± | 58.38 | 4 041 | ± | 170.7 |

As for the zooming phase, the full query implementation is always the slowest while the single queries are slower than the 2D queries. Moreover the zooms are the fastest on R2 and vR trees. While this trace triggers R3 when built with the vR algorithm, this happens rather late and therefore does not influence the tree structure too much.

### 7.3. Storage usage

We see in Figure 14 that the size for the classic SHT would be proportional to the square of the number of attributes. Sizes for the eSHT and R-SHTs are aligned with the raw size of the intervals that are being stored, with a slight overhead for the header information. Therefore, the data structure no longer wastes space when stored in external memory. Because of the SHT's inefficient serialization, we could not fit it in external memory for traces producing more than $2.1 \times 10^6$ attributes, whereas the other implementations executed properly until $4.5 \times 10^6$.

### 7.4. Tree Depth

We see in Figure 15 that the depth of the classic SHT would be proportional to the number of attributes. Depths for the eSHT and R-SHTs are closer to those of packed and balanced trees. As the latest branch is kept in memory before being serialized, this results in substantial memory savings during the build process. Moreover, we no longer have long empty branches as we had in Figure 3.

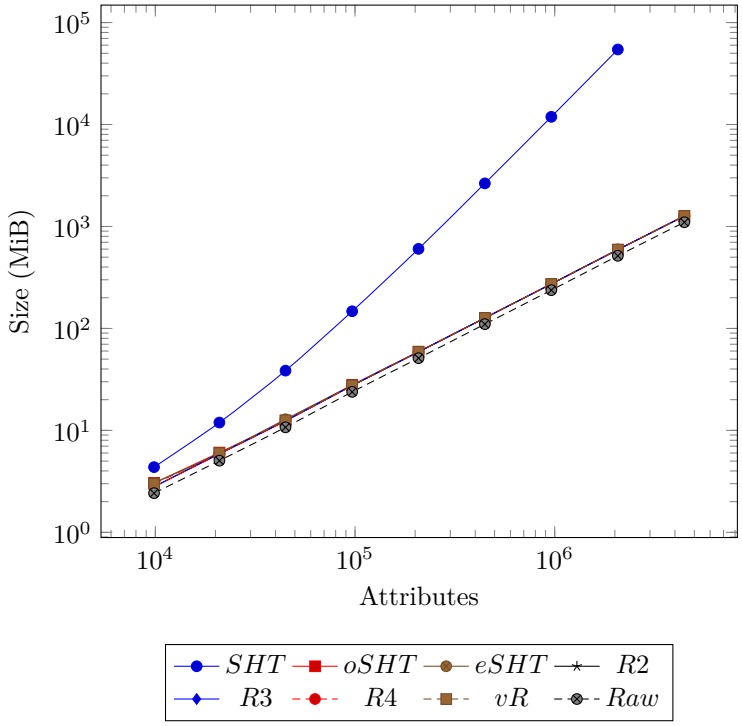



Fig. 14: Comparison of external memory usage for different SHT variants

### 7.5. Single Query

While the eSHT does not make queries significantly faster, Figure 16 shows how the reduction of node overlap along with increasing the R-Tree buffer depth reduces the single query search time.

Moreover, the depth increase of the R-buffer with the variable height implementation is clearly visible, with query times similar to $R = 2$ until $Attributes \approx 2 \times 10^6$ then close to $R = 3$ for $Attributes = 4 \times 10^6$ and on.

### 7.6. Full Query

In Figure 17 , the move from SHT to eSHT does not impact the full query performance, as the same number of nodes remains to be searched. Meanwhile, the effect of the R-SHT, grouping intervals with the same attributes, at the cost of similar time range groupings, significantly reduces the full query performance.

### 7.7. 2D Query

We benchmarked the 2D query on its ability to extract a sparse set of 100 keys on a set of 2000 timestamps. We compare how much time it would have taken to query the same data with single or full queries. We do not represent the full queries on the vR tree, as we saw in the previous section that they are slower than on the eSHT.

We see in Figure 18 that the 2D query approach is faster than repeated single queries on both eSHT and vR trees. Moreover, both types of queries are faster on the vR tree.

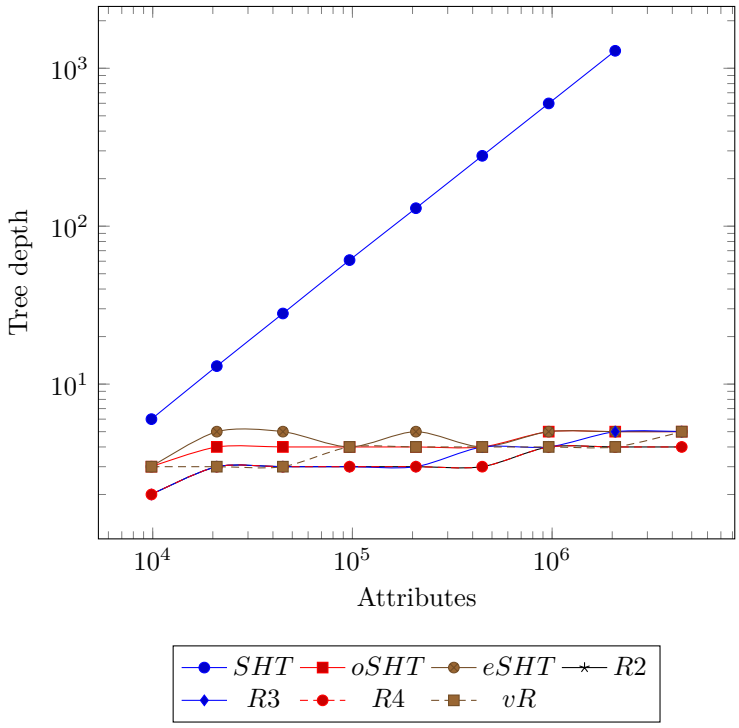



Fig. 15: Comparison of tree depth with different SHT variants

### 7.8. Build Time

As the move from SHT to eSHT strongly reduces the number of nodes that need to be created and written to external media, the eSHT build is much faster than the R-SHT's, as can be seen in Figure 19. However, the R-SHT optimizations resort to sorting a list of intervals as long as the R-Tree buffer's capacity, which increase the build time.

## 8. CONCLUSION

In this paper, we have presented two new evolutions of the State History Tree (SHT) data structure.

The first variant, the enhanced State History Tree (eSHT), addresses the issue of scalability when systems with many threads are analyzed. It focuses on near-optimal storage usage and tree depth, both of which impact the build time of the tree.

The second variant, R-SHT, focuses on query performance, which the shift from SHT to eSHT did not significantly improve. R-SHT uses R-Tree techniques to more efficiently organize the data in the tree structure, so that it is faster to retrieve. Moreover, we suggest a data-driven method to adjust the level of optimization so that query times remain below a threshold, regardless of the number of attributes.

We modeled the data structures' behaviors, showing the gains on query performance with different levels of optimization. We put the new implementations to trial by analysing highly parallel traces to show the benefits on query performance. Finally, we studied the implementation of the data structure in the Trace Visualization and Analysis software Trace Compass, finding significant gains (at least 7×) in the time it took to retrieve and render complex views.

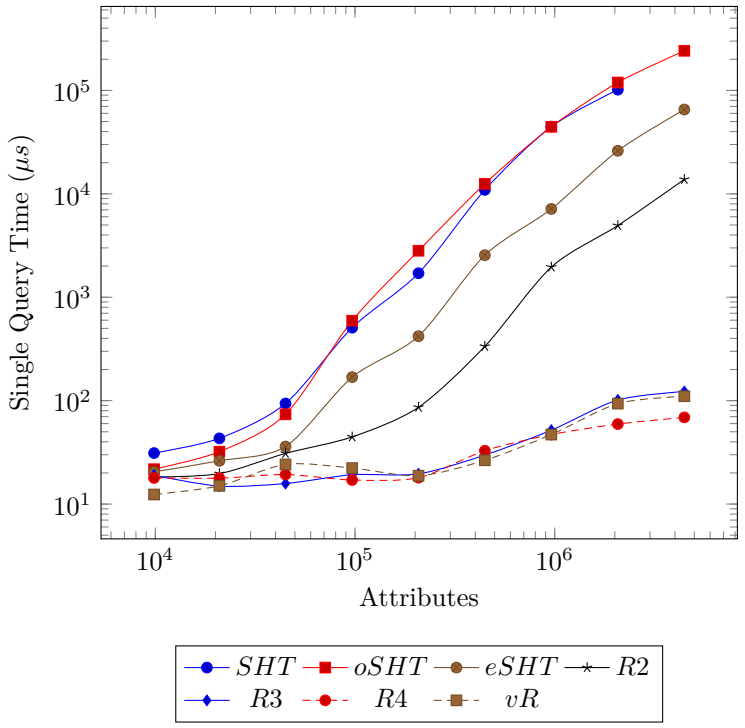



Fig. 16: Comparison of single query performance with different SHT variants

We find that the R-SHT is up to three orders of magnitude faster for single queries and 2D queries than the SHT and eSHT for traces with more than one million attributes. However, it is less efficient on full queries, but we showed that using full queries to populate views is less efficient than the others.

We believe that these new structures, especially the R-SHT, will enable the analysis and visualization of traces to scale to highly parallel systems with millions of threads or cores. Moreover, vR solves the problem of determining the optimization level, by changing it on the fly according to the data properties.

Future work could use Mip-Mapping techniques to store a summary of the more detailed information stored in the structure, as to reduce the computation required at less zoomed-in levels.

## References

Elke Achtert, Hans-Peter Kriegel, and Arthur Zimek. 2008. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proceedings of the 20th International Conference on Scientific and Statistical Database Management (SSDBM '08)*. Springer-Verlag, Berlin, Heidelberg, 580–585. DOI:http://dx.doi.org/10.1007/978-3-540-69497-7_41

L Arge, KH Hinrichs, J Vahrenhold, and JS Vitter. 2002. Efficient Bulk Operations on Dynamic R-Trees1. *Algorithmica* 33 (2002), 104–128.

Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal* 5, 4 (Dec. 1996), 264–275. DOI:http://dx.doi.org/10.1007/s007780050028

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Rec.* 19, 2 (May 1990), 322–331. DOI:http://dx.doi.org/10.1145/93605.98741

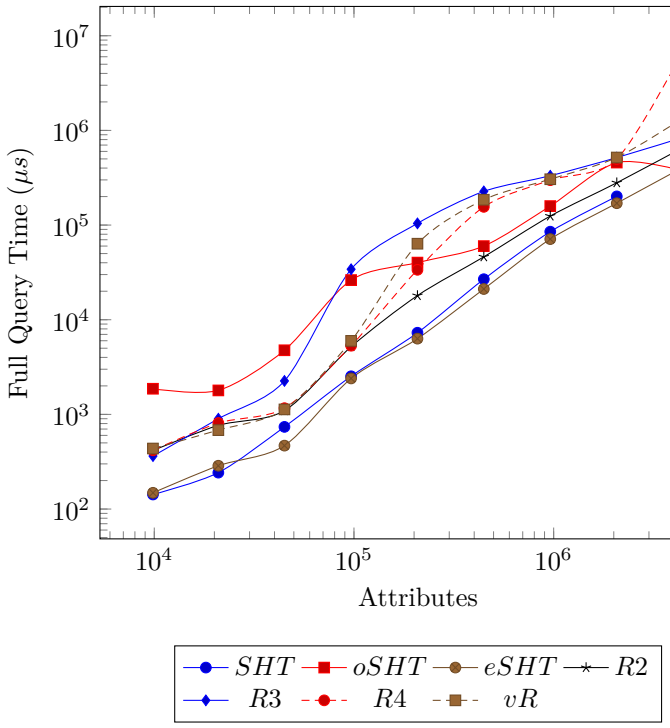



Fig. 17: Comparison of full query performance with different SHT variants

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA, Chapter 10.

Tim Bird. 2009. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*. Citeseer, 47–54.

Laurynas Biveinis, Simonas Šaltenis, and Christian S Jensen. 2007. Main-memory operation buffering for efficient R-tree update. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 591–602.

Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis. 2002. Revisiting R-Tree Construction Principles. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems (ADBIS '02)*. Springer-Verlag, London, UK, UK, 149–162. http://dl.acm.org/citation.cfm?id=646046.676614

Anthony Chan, William Gropp, and Ewing Lusk. 2008. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming* 16, 2-3 (2008), 155–165.

Li Chen, Rupesh Choubey, and Elke A. Rundensteiner. 1998. Bulk-insertions into R-trees Using the Small-tree-large-tree Approach. In *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (GIS '98)*. ACM, New York, NY, USA, 161–162. DOI:http://dx.doi.org/10.1145/288692.288722

Rupesh Choubey, Li Chen, and Elke A. Rundensteiner. 1999. GBI: A Generalized R-Tree Bulk-Insertion Strategy. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases (SSD '99)*. Springer-Verlag, London, UK, UK, 91–108. http://dl.acm.org/citation.cfm?id=647226.719080

Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. DOI:http://dx.doi.org/10.1145/356770.356776

Thomas H Cormen. 2009. *Introduction to algorithms* (3 ed.). MIT Press, Chapter 14.

Antonio Corral, Michael Vassilakopoulos, and Yannis Manolopoulos. 2001. The Impact of Buffering on Closest Pairs Queries Using R-Trees. In *Proceedings of the 5th East European Conference on Advances in Databases and Information Systems (ADBIS '01)*. Springer-Verlag, London, UK, UK, 41–54. http://dl.acm.org/citation.cfm?id=646045.676587

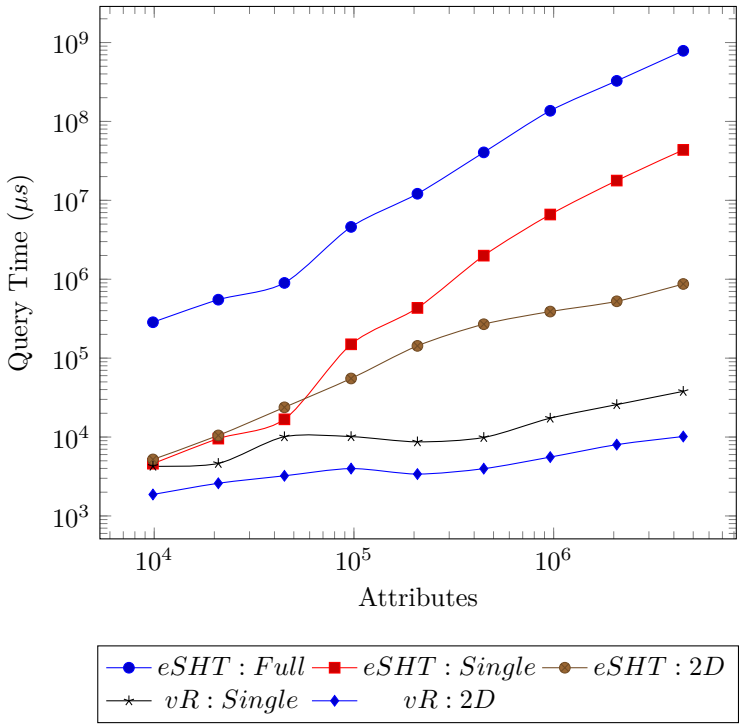



Fig. 18: Comparison of full, single and 2D queries to extract the same data on eSHT and vR-SHT

Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. 2012. An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011*. Springer, 37–48.

Mathieu Côté and Michel R. Dagenais. 2016. Problem Detection in Real-Time Systems by Trace Analysis. *Advances in Computer Engineering* 2016 (2016), 12. DOI:http://dx.doi.org/10.1155/2016/9467181 Article ID 9467181.

Mathieu Desnoyers and Michel Dagenais. 2006a. OS tracing for hardware, driver and binary reverse engineering in Linux. *CodeBreakers Journal* 1, 2 (2006).

Mathieu Desnoyers and Michel R Dagenais. 2006b. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *OLS (Ottawa Linux Symposium)*, Vol. 2006. Citeseer, 209–224.

David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. 2005. The Skip Quadtree: A Simple Dynamic Data Structure For Multidimensional Data. (June 2005). http://www.ics.uci.edu/~eppstein/pubs/EppGooSun-SoCG-05.pdf Presentation at the 21st ACM Symp. on Computational Geometry, Pisa, June 2005.

Naser Ezzati-Jivan and Michel R Dagenais. 2012. A stateful approach to generate synthetic events from kernel traces. *Advances in Software Engineering* 2012 (2012), 6.

R. A. Finkel and J. L. Bentley. 1974. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4, 1 (March 1974), 1–9. DOI:http://dx.doi.org/10.1007/BF00288933

M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. Wylie. 2007. *Scalable Collation and Presentation of Call-Path Profile Data with CUBE*. NIC series, Vol. 38. John von Neumann Institute for Computing, Jülich, 645–652. http://juser.fz-juelich.de/record/58173 Record converted from VDB: 12.11.2012.

Inc Google. 2013. Web Tracing Framework. (2013). http://google.github.io/tracing-framework/

Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, Vol. 14. ACM, New York, NY, USA, 47–57. DOI:http://dx.doi.org/10.1145/602259.602266

Sariel Har-Peled. 2011. *Geometric approximation algorithms*. Vol. 173. American mathematical society Providence. 348–354 pages.

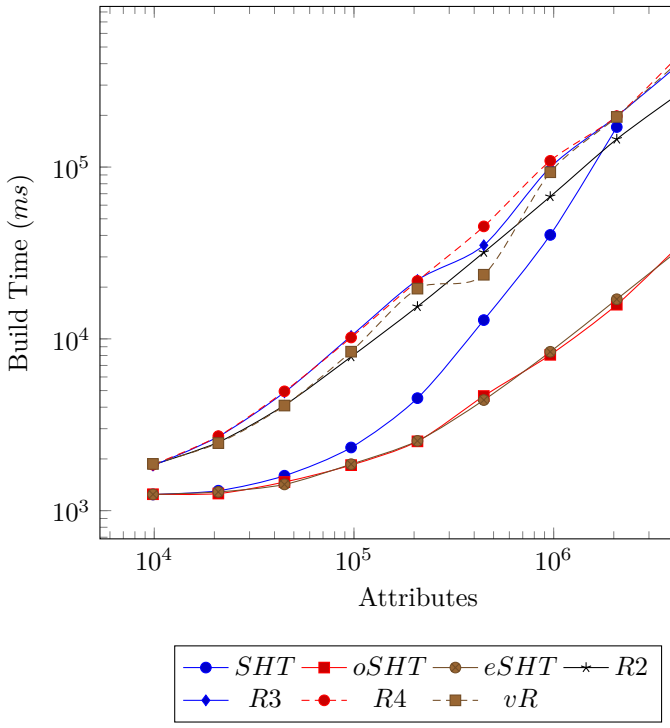



Fig. 19: Comparison of build time with different SHT variants.

Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 500–509. http://dl.acm.org/citation.cfm?id=645920.673001

A. Korotkov. 2012. A new double sorting-based node splitting algorithm for R-tree. *Programming and Computer Software* 38, 3 (May 2012), 109–118. DOI:http://dx.doi.org/10.1134/S0361768812030024

G. G. Lai, D. Fussell, and D. F. Wong. 1993. HV/VH Trees: A New Spatial Data Structure for Fast Region Queries. In *Design Automation, 1993. 30th Conference on*. 43–47. DOI:http://dx.doi.org/10.1109/DAC.1993.203917

Scott T. Leutenegger and Mario A. López. 2000. The Effect of Buffering on the Performance of R-Trees. *IEEE Trans. on Knowl. and Data Eng.* 12, 1 (Jan. 2000), 33–44. DOI:http://dx.doi.org/10.1109/69.842248

J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. University of California Press, Berkeley, Calif., 281–297. http://projecteuclid.org/euclid.bsmsp/1200512992

Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. 2010. *R-trees: Theory and Applications*. Springer Science & Business Media.

Alexandre Montplaisir, Naser Ezzati Jivan, Florian Wininger, and Michel Dagenais. 2013. Efficient Model to Query and Visualize the System States Extracted from Trace Data. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. 219–234. DOI:http://dx.doi.org/10.1007/978-3-642-40787-1_13

A. Montplaisir-Goncalves, N. Ezzati-Jivan, F. Wininger, and M.R. Dagenais. 2013. State History Tree: An Incremental Disk-Based Data Structure for Very Large Interval Data. In *Social Computing (SocialCom), 2013 International Conference on*. 716–724. DOI:http://dx.doi.org/10.1109/SocialCom.2013.107

Aouatef Mrad, Samatar Ahmed, Sylvain Hallé, and Éric Beaudet. 2013. Babeltrace: A collection of transducers for trace validation. In *Runtime Verification*. Springer, 126–130.

Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. 2007. Developing Scalable Applications with Vampir, VampirServer and Vampir-

Trace.. In *PARCO* (2009-02-16) *(Advances in Parallel Computing)*, Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters (Eds.), Vol. 15. IOS Press, 637–644. http://dblp.uni-trier.de/db/conf/parco/parco2007.html#MullerKJLBMN07

Mario A. Nascimento and Jefferson R. O. Silva. 1998. Towards Historical R-trees. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*. ACM, New York, NY, USA, 235–240. DOI:http://dx.doi.org/10.1145/330560.330692

Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. 2009. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 696 – 710. DOI:http://dx.doi.org/10.1016/j.jpdc.2008.09.001 Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. DOI:http://dx.doi.org/10.1145/2400682.2400712

Loïc Prieur-Drevon, Raphaël Beamonte, Naser Ezzati, and Michel Dagenais. 2016. Enhanced State History Tree (eSHT) : a Stateful Data Structure for Analysis of Highly Parallel System Traces. In *2016 IEEE International Congress on Big Data*. IEEE. DOI:http://dx.doi.org/10.1109/BigDataCongress.2016.19

Gururaj S. Rao. 1978. Performance Analysis of Cache Memories. *J. ACM* 25, 3 (July 1978), 378–395. DOI:http://dx.doi.org/10.1145/322077.322081

Nick Roussopoulos and Daniel Leifker. 1985. Direct Spatial Search on Pictorial Databases Using Packed R-trees. *SIGMOD Rec.* 14, 4 (May 1985), 17–31. DOI:http://dx.doi.org/10.1145/971699.318900

Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518. http://dl.acm.org/citation.cfm?id=645914.671636

Yufei Tao and Dimitris Papadias. 2001. The mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome*.